Systematic Searches for Global Multiprocessor Real-Time Scheduling

Olivier Buffet, Liliana Cucu-Grosjean

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Systematic Searches for Global Multiprocessor Real-Time Scheduling

Olivier Buffet — Liliana Cucu-Grosjean

N° 7386

19 septembre 2010

Embedded and Real Time Systems

ISSN 0249-6399 ISRN INRIA/RR--7386--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
NANCY - GRAND EST

Systematic Searches for Global Multiprocessor Real-Time Scheduling

Olivier Buffet, Liliana Cucu-Grosjean

Theme : Embedded and Real Time Systems
Algorithmics, Programming, Software and Architecture
Équipes-Projets Maia et Trio

Rapport de recherche n° 7386 — 19 septembre 2010 — 17 pages

Abstract: In this paper we address the problem of global real-time periodic scheduling on homogeneous multiprocessor platforms. A number of theoretical results have been obtained in the field of real-time systems, but mainly focusing on properties of specific algorithms in uniprocessor settings. The multiprocessor case has been considered only recently, with few resolution techniques proposed and experimented with up to now. In this paper we discuss several systematic search algorithms—exploring different search spaces—that exploit various features of the problem. These approaches are then evaluated experimentally on numerous randomly generated problems. This work shows (1) how two heuristic approaches can solve most (feasible and unfeasible) problems in no time, and (2) how to improve a state of the art algorithm by looking at jobs' *laxities* and by focusing the search on bottlenecks. We also discuss limitations of the proposed solvers and future work.

Key-words: global realtime periodic scheduling; multiprocessor scheduling; systematic search

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Recherches systématiques pour l'ordonnancement global multiprocesseur temps-réel

Résumé : Dans cet article nous nous intéressons au problème de l'ordonnancement périodique global temps-réel sur plateformes multiprocesseurs homogènes. De multiples résultats théoriques ont été obtenus dans le domaine des systèmes temps-réel, mais se focalisant principalement sur des propriétés d'algorithmes spécifiques dans des cadres multiprocesseurs. Le cas multiprocesseur n'a été considéré que récemment, seules peu de techniques de résolution ayant été proposées et validées expérimentalement jusqu'ici. Dans cet article nous discutons de plusieurs algorithmes de recherche systématique — explorant des espaces de recherche différents — qui exploitent des caractéristiques diverses du problème. Ces approches sont ensuite évaluées expérimentalement sur de nombreux problèmes générés aléatoiremenent. Ce travail montre (1) comment deux approches heuristiques peuvent résoudre la plupart des problèmes en très peu de temps, et (2) comment améliorer un algorithme de l'état de l'art en considérant la *laxité* des jobs et en focalisant la recherche sur les goulots d'étranglement. Nous discutons aussi des limitations des solveurs proposés et des travaux futurs.

Mots-clés : ordonnancement périodique global temps-réel; ordonnancement multiprocesseur; recherche systématique

Contents

1 Introduction

Requests in real-time environments are often of a recurring nature. Such systems are typically modeled as finite collections of simple, highly repetitive activities (e.g., tasks, messages) and they involve the sharing of one or more resources among various processes. Moreover, for these systems the logical results of the computation, as well as the time at which these results are produced, are important. Therefore, they are modeled as finite collections of repetitive tasks, each of which generates jobs that must be executed within some time interval. The mechanism that decides which job[s] should be executed at each time instant and on what processor is the *scheduling algorithm*. When there is at least one schedule satisfying all constraints of the system, the system is said to be *feasible*.

Uniprocessor real-time scheduling problems are well studied since the seminal paper of Liu and Layland [10]. The literature considering scheduling algorithms and feasibility tests for uniprocessor scheduling is tremendous. In contrast for *multiprocessor* platforms the problem of meeting timing constraints is a relatively new research area.

In the design of scheduling algorithms for multiprocessor environments, one can distinguish between at least two distinct approaches. In *partitioned scheduling*, all jobs generated by a task are required to execute on the *same* processor. *Global scheduling*, by contrast, permits *task migration* (i.e., different jobs of an individual task may execute upon different processors) as well as *job migration* (an individual job that is preempted may resume execution upon a processor different from the one upon which it had been executing prior to preemption). In this work we consider global preemptive scheduling.

From a theoretical and practical point of view we can distinguish between at least three kinds of multiprocessor platforms (from less general to more general). We deal with *identical processors* if all processors are identical, in the sense that they have the same computing power. By contrast, in the case of *uniform* processors, each processor P_j is characterized by its own computing capacity: a job that executes on processor P_j of computing capacity s_j for t time units completes $s_j \times t$ units of execution. Finally in the case of *heterogeneous processors* there is an execution rate $s_{i,j}$ associated with each job-processor pair: a job J_i that executes on processor P_j for t time units completes $s_{i,j} \times t$ units of execution. The heterogeneous processors model dedicated processors (e.g., $s_{i,j} = 0$ means that P_j cannot serve job J_i).

Related research. The problem of scheduling periodic task systems on several processors was originally studied in [9]. Recent studies provide a better understanding of that scheduling problem and provide first solutions. E.g., [2] and [5] present a categorization of real-time multiprocessor scheduling problems.

The difficulty when one studies multiprocessor scheduling comes from the fact that uniprocessor feasibility results do not always hold for multiprocessor scheduling. For instance the synchronous case (i.e., considering that all tasks start their execution synchronously) is not a worst case for all asynchronous situations upon multiprocessors [7]. Therefore most of the results indicate that real-time multiprocessor scheduling problems are typically not solved by applying straightforward extensions of techniques used for solving similar uniprocessor problems [6].

1.1 Contribution of this paper

In this paper, we look at various methods for finding feasible schedules for systems with identical processors, from simple heuristics to complex AI search algorithms. A similar approach has already been used for multiprocessor real-time scheduling but in the partitioned case [11]. To our best knowledge, using search algorithms to solve the global multiprocessor real-time scheduling problem is a new research direction, initiated by [4] with a focus on constraint satisfaction techniques. The main contributions of this work are (1) how two heuristic approaches can solve most (feasible and unfeasible) problems in no time, and (2) how to improve the second CSP encoding (CSP2) proposed in [4]—by looking at jobs' *laxities* and by focusing the search on bottlenecks—allowing now for solving more difficult instances.

1.2 Organization of the paper

The paper is organized as follows. The next section provides the model of tasks and its associated notations, and briefly present notions on constraint satisfaction problems before defining the constraints of a multiprocessor global real-time scheduling problem. Then Section 3 is devoted to presenting the various algorithms we introduce or improve on, by increasing order of complexity. The experiments we have conducted are then described and analyzed in Section 4. Finally, the results obtained as well as future work are discussed in the last section.

2 Background

2.1 Model and Associated Notations

We consider the global preemptive[1] scheduling of periodic tasks on identical processors. A task system τ is composed by n tasks and each task is characterized by a 4-tuple (O_i, C_i, D_i, T_i) where:

O_i is the *offset* of task τ_i, i.e., the time instant when the first availability interval starts with respect to $t = 0$;

C_i is the *worst-case execution time* (WCET) of task τ_i;

T_i is the *period* of task τ_i. Let $T_{max} = \max\{T_1, T_2, \cdots, T_n\}$;

D_i is the *deadline* of task τ_i, i.e., each job k generated at time $O_i + (k-1)T_i$ must finish its execution before $D_i + O_i + (k-1)T_i$ (implying that $C_i \leq D_i$).

In this paper we consider the case of constrained deadline task systems, i.e., $D_i \leq T_i, \forall i \leq n$. The time being discrete, all these parameters have integer values.

A solution of the *Multiprocessor Global Real-Time Scheduling* (MGRTS) problem for any task system $\tau = \{\tau_1, \ldots, \tau_n\}$ and any set of m processors $\{P_1, \ldots, P_m\}$ is defined by a *schedule* $\sigma : \mathbb{N} \to \{0, 1, \ldots, n\}^m$ where, at any time t,
$\sigma(t) \stackrel{\text{def}}{=} (\sigma_1(t), \sigma_2(t), \ldots, \sigma_m(t))$ with
$\sigma_j(t) \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if there is no task scheduled on } P_j \\ & \text{at instant } t; \\ i, & \text{if } \tau_i \text{ is scheduled on } P_j \text{ at instant } t; \end{cases}$
$\forall 1 \leq j \leq m$.

[1] A task is preemptive if it can be interrupted at any time.

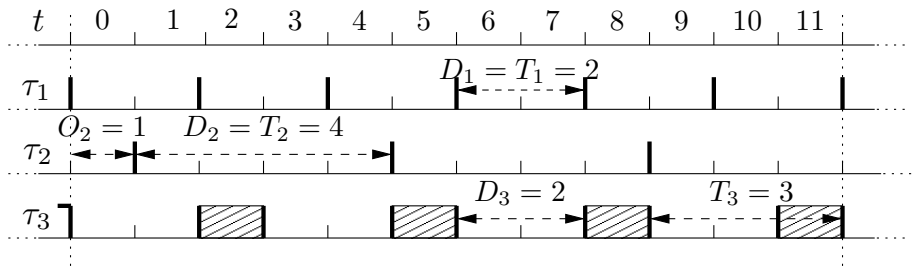

Figure 1: Representation of the availability intervals of example 1's tasks during one hyperperiod (with $O_1 = O_3 = 0$)

We consider that parallelism is forbidden, i.e., a task is scheduled at any time instant on at most one processor. Moreover, any processor can execute at any time instant at most one task.

A feasible schedule is a schedule with all deadlines met for all jobs of all tasks. Therefore in a feasible schedule and from the definition of the deadline, a job k of a task τ_i must be scheduled within its *availability interval*:

$$I_{i,k} = [O_i + (k-1)T_i, D_i + O_i + (k-1)T_i).$$

Example 1. *In the remainder of the paper, we use the following running example:* $m = 2$*,* $n = 3$*, and the tasks are defined by:*

τ_i	O	C	D	T
τ_1	0	1	2	2
τ_2	1	3	4	4
τ_3	0	2	2	3

Let T_H be the least common multiple of all periods: $T_H = lcm(T_1, \cdots, T_n)$. Since the same pattern of availability intervals repeats every T_H time instants for any task set, T_H is the *hyperperiod* of the availability intervals. In Example 1, the hyperperiod is $T_H = 12$ and the pattern of availability intervals is the one shown on Figure 1.

2.2 Constraint Satisfaction Problems

As we will see in the next section, MGRTS problems can be formalized as *Constraint Satisfaction Problems* (CSPs) [12, 8], although not necessarily in the same way as more classical scheduling problems [1]. A CSP is defined by a set of *variables* $\mathcal{X}_1, \cdots, \mathcal{X}_p$—each defined on a *domain* $\mathcal{D}_i$—and a set of *constraints* $\mathcal{C}_1, \cdots, \mathcal{C}_q$ involving one or several variables. A *state* is a partial or complete assignment of values to variables. Solving a CSP means either (1) finding a solution, i.e. a complete state satisfying all constraints, or (2) making sure that no valid/consistent solution exists.

A real-world problem can often be formalized as a CSP in various ways and various resolution strategies can be envisioned. The difficulty is then to find the best combination.

2.3 Formalizing an MGRTS Problem as a CSP

A first point is that we are dealing with a satisfaction problem, not an optimization one.

A second important point is the periodicity property of any feasible schedule of constrained deadline task systems proved in [3]. This property states that, for a constrained deadline periodic system, and because of the existence of an availability intervals pattern, there exists a feasible schedule for all jobs of all tasks if and only if there exists a feasible finite schedule within an interval of length T_H. This property holds in our setting, allowing us to focus our effort on periodic solutions. This guarantees that a finite number of variables is sufficient, which is a prerequisite to translate the MGRTS problem to a constraint satisfaction problem.

An MGRTS problem is then the problem of finding a feasible *periodic* schedule for a periodic task system $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$ upon m identical processors $P_1, P_2, \cdots, P_m$. Each task τ_i is given by (O_i, C_i, D_i, T_i) and we have $D_i \leq T_i, \forall i \leq n$. Finding a feasible schedule implies that the following conditions are met:

C1 Each job of task τ_i is scheduled within its corresponding availability interval.

C2 At time t, on processor P_j, at most one task is running.

C3 At time t, task τ_i runs on at most one processor.

C4 Task τ_i should last exactly C_i during each availability interval.

In this paper, only the CSP2 approach (Sec. 3.3) formalizes the MGRTS problem precisely in this way, the search space being finite (because we have a finite number of variables, each with a finite domain). The other approaches explore smaller search spaces, but still rely on this formalization to validate their solutions by expanding full candidate schedules.

2.4 Systematic Search Algorithms

We will always consider finite search spaces, allowing us to focus on systematic generate-and-test procedures (we do not consider local search algorithms). For a given CSP, such a procedure can take the form of a depth-first search starting from the "empty-assignment" state and trying all possible value assignments for variable $\mathcal{X}_i$ when at depth i (the algorithm backtracks when a constraint is violated). Since we use hand-made ad hoc solvers rather than generic CSP solvers, describing a particular solver will require (1) defining the problem formalization and (2) giving details regarding the search such as:

- *how the variables are ordered* (to prune the search space more efficiently);
- *how the values are ordered* in each variable's domain (so as to find a consistent solution earlier);
- *which constraints are added* (to reduce the size of the search space if this does not make the problem unsolvable).

3 Some Solution Techniques

This section first discusses generic ideas that will be used in the solution techniques studied in this paper. Then it presents some heuristic approaches to find feasible schedules before describing more advanced search algorithms.

3.1 Preliminary Remarks

3.1.1 Necessary Condition

Let us denote $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ the *utilization factor* for a given problem. It is usual to first check whether a scheduling problem is feasible by verifying that this utilization factor is smaller than (or equal to) the number of available processors: $U \leq m$.

Here, we refine this first necessary condition by observing that m is an upper-bound on the number of tasks that can run simultaneously. If, at some time step t, the number m_t of currently "active" availability intervals is less than the number of processors m, then U should not be compared with m, but with the *average number of simultaneously executable tasks* m':

$$m' = \frac{1}{T_H} \sum_{t=0}^{T_H-1} \max(m, m_t).$$

We will refer to this second test ($U \leq m'$) as the *Necessary Condition* (NC) algorithm. It will be employed before any solver to avoid unnecessary computations.

More complex necessary conditions could be designed, but one should be careful to keep such a pre-processing step time and memory efficient. We have conducted experiments with an algorithm that looks at the usage of processors over sliding time windows, but its time complexity ended up being too large considering the small gain in efficiency.

3.1.2 Testing a Solution

In the present paper, some algorithms work directly with an expanded schedule, detailing which job is running at which time step on which processor, while other algorithms work on a simpler representation. In all cases, a solution sol—whatever its formalization—is tested by developing the corresponding schedule. The develop-and-test function will be noted $test(sol)$.

An important question is how to properly turn a solution sol into a schedule on a time interval of length T_H that can repeat over time. The answer depends on the algorithm at hand. We therefore postpone it to the description of the solution techniques.

3.2 Looking for Fixed Priority Assignments

An approach that has been extensively studied in the uniprocessor case is that of assigning a *fixed priority* (FP) to each task (a non-negative integer) prior to execution. Then, at any time during execution, non-completed jobs are assigned to processors by priority order. As can be expected, there exist feasible problems with no feasible FP assignment (e.g. Example 1).

Testing an FP assignment can be done by filling the developed schedule one task after another—sorted by priority—one availability window at a time. The resulting schedule will naturally be repeatable infinitely often.

3.2.1 Fixed Priority with No Search

To find a satisfying FP assignment, a first solution is to use heuristics such as:

0. **none:** no heuristic;

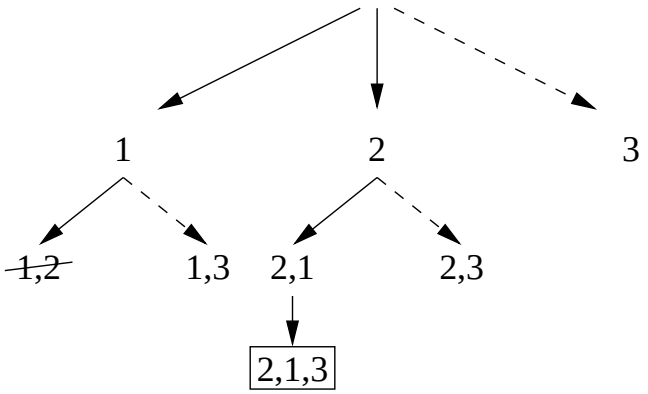

Figure 2: An example tree of partial FP orderings for $n = 3$ tasks. Dashed arrows indicate non-visited branches. $test(\dots)$ returns a success for all visited nodes except $(1, 2)$.

1. **RM:** tasks with smallest period first (Rate Monotonic);
2. **DM:** tasks with smallest deadline first (Deadline Monotonic);
3. **T-C:** tasks with smallest $T - C$ value first;
4. **D-C:** tasks with smallest $D - C$ value first.

Ties are broken randomly, so that heuristic 0 gives a random FP assignment. Some FP assignment rules (RM and DM for example) are known to be optimal under certain conditions, requiring in particular that only one processor is used and that tasks are synchronous ($\forall i,\ O_i = 0$). There is currently no such result in the multiprocessor case. From now on, these first five approaches will be referred to as NS(u) (for "No Search"), with $u \in 0..4$.

3.2.2 Fixed Priority with Systematic Search

A logical improvement about the NS approaches is to perform a search in the space of FP assignments. This paper only looks at systematic search algorithms, even if the size of the search spaces grows quickly with the number of tasks: in particular, there are $n!$ complete FP assignments.

Here, we propose a depth-first search in the tree of all partial FP assignments (see Figure 2). In this tree, a node is an ordered list of tasks (the root being the empty list), and children are obtained by adding a task at the end of the list (and internal node at depth d having $n - d$ children). Two interesting properties that facilitate this search procedure are the possibilities:

- to incrementally build the full schedule as new tasks are added: each new task has no impact on higher priority tasks; and
- to prune a branch B as soon as a child node (B, c) is proved unfeasible (because any other sub-branch will add c after B, as on Fig. 2 with B=(1) and c=2).

Various instances of this search can be distinguished depending on the ordering of children. We will note FPS(u) the search where a node's children are sorted according to heuristic $u \in 0..4$ (see previous section).

3.3 Exhaustive Search

Although we have experimented with both exhaustive approaches introduced in [4] (CSP1 and CSP2 both perform a search that will find a schedule if one exists), we only describe the second one, CSP2, proposing some significant improvements.

This problem exhibits interesting characteristics. As we will see, in this problem the variables are naturally organized as a 2-dimensional array (the dimensions being "time" and "processor"). The constraints will be very local, so that it makes a lot of sense to loop through the variables in a pre-determined order.

These various positive aspects are counterweighed by the fact that our problem is cyclic. Indeed, most classical CP scheduling techniques [1] do not apply in this setting. For example, some of them (e.g., "Edge-Finding") assume that tasks can be ordered between a start and an end point, what does not make sense when there is no start and no end.

3.3.1 Original CSP2

Variables In this CSP, we use one n-ary variable $x_j(t)$ per processor j and time step t—where $t \in 1..T_H$—indicating which task τ_i (-1 if no task) runs on processor j at time t:

$$x_j(t) = \begin{cases} i & \text{if } \tau_i \text{ on } P_j \text{ at } t; \\ -1 & \text{otherwise.} \end{cases}$$

We therefore have $m * T_H$ variables, each taking one of $n+1$ possible values ($D_j(t) = \{-1, 1, 2, \cdots, n\}$). However T_H, which depends on the periods of the various tasks, may take large values. In the worst case, T_H is the least common multiplier of the $T_{\max}$ first integers ($lcm(1, \ldots, T_{\max})$), a sequence whose behavior is asymptotic with $e^{T_{\max}}$ when $T_{\max} \to \infty$ [13].[2]

By reducing the number of variables and because some constraints are induced by the choice of the variables and their domains, the problem to solve is defined only by conditions (C1), (C3) and (C4).

Constraints With these variables, we define the following constraint satisfaction problem that we denote by CSP2:

$$x_j(t) \neq i,\ \forall t \notin I_{i,1} \cup \cdots \cup I_{i,\frac{T_H}{T_i}}; \tag{1}$$

$$x_j(t) = x_{j'}(t) \Leftrightarrow x_j(t) = -1; \tag{2}$$

$$\sum_{t \in I_{i,k}} \sum_{j} \delta(i, x_j(t)) = C_i,\ \forall k \in 1..\frac{T_H}{T_i}; \tag{3}$$

where $\delta(a, b) = 1$ if $a = b, 0$ if $a \neq b$.

[2] `http://www.research.att.com/~njas/sequences/A003418`

Search Strategy As explained previously, efficiently solving CSPs with a backtracking algorithm requires carefully controlling the search. We now explain the choices made in CSP2's implementation (not detailing how constraints (1)–(3) are implemented).

Ordering the Variables — Considering the variables, the main decision is to order them first in chronological order and to work time-step by time-step. Even if time is cyclic (periodic modulo T_H) in our problem, this ordering is beneficial as it ensures that new decisions are taken given the knowledge of most past events. In the present case, as only identical processors are considered, all processor orderings are equivalent. So we just order them according to their id number.

Ordering the Values — The values correspond to the possible tasks $(1..n)$ or to the absence of any task (-1). As previously, we will consider heuristics 0 to 4 to order the jobs (see NS(u) and FPS(u) approaches), so that we will have several versions CSP2(u).

In addition (this is a novelty over [4]), we also consider a fifth version (CSP2(5)) using a dynamic ordering: At a given time step, jobs will be ordered by increasing laxity (*least laxity first* heuristic (LLF)). The *laxity* (or *slack time*) of a task τ_i at time t is defined as:

$$lax(i,t) = \begin{cases} \alpha(i,t) - \beta(i,t) & \text{if } t \in I_{i,1} \cup \cdots \cup I_{i,\frac{T_H}{T_i}} \\ +\infty & \text{otherwise,} \end{cases}$$

where $\alpha(i,t)$ is the time left before the next deadline and $\beta(i,t)$ is the number of time units of task i that need to be scheduled before this deadline. Note that the laxity can be computed at a low cost while incrementally building the schedule.

Adding Constraints — In fact, the following two constraints—which exploit symmetries—are not explicitely added to the CSP model, but are used directly in the design of the search procedure:

- Considering the "no task" value, it does not make sense to leave a processor idle at time t if some task can run on it. Thus, a first rule is: *the "no task" value should be used only when no tasks are available for running.*
- One can also observe that, at any time step t, all permutations of tasks on processors are equivalent. The number of value assignments for time step t can therefore be divided by up to $m!$ by applying a second rule: *tasks and processors should be considered in ascending order only, i.e.:*

$$(j < j') \Leftrightarrow (x_j(t) \leq x_{j'}(t)). \tag{4}$$

3.3.2 Improving on CSP2

Using Slack It is also possible to backtrack early by looking at the laxity of the tasks: *the search can backtrack from time-step t to $t-1$ as soon as the laxity of some task τ_i is non-positive, which means that there is not enough time to complete the corresponding job.* Putting it differently, the following constraint should be satisfied at time t:

$$lax(i,t) \geq 0,\ \forall i \in 1..n.$$

Versions of CSP2(u) exploiting this "slack constraint" will be denoted with the letter 's': CSP2(u)s.[3]

Reversing a Problem With CSP2(u), waiting a few minutes without solving the problem probably means that the search does not go past some maximum horizon point $t_{\max}$ which constitutes a bottleneck either because of inappropriate past decisions or because of a local set of constraints that make the problem unfeasible. In both cases, a good idea is to focus the search on this bottleneck. If searching forward, we should start close to $t_{\max}$ (to focus), but not too close (otherwise we may miss the difficulty). A simple solution is to start right after $t_{\max}$ and solve the *backward* scheduling problem. Indeed, a GMRTS problem is identical if you reverse the time arrow. This is achieved by mapping each task τ_i to an reversed task τ'_i which only differs by its offset $O'_i = (T_i - D_i - O_i)\%T_i$ (value obtained by first observing that $O_i + O'_i = T_i + D_i$, then constraining $O'_i \in (0, T_i)$).

Versions of CSP2(u) exploiting this "reversing bottlenecks" approach will be denoted the letter 'r': CSP2(u)r. It can be freely combined with the slack constraint, giving rise to the CSP2(u)sr version.

4 Experiments

For these experiments, all algorithms have been implemented in C++ except CSP1, which uses a simple model and relies on a generic CSP solver, `Choco` [14]. Each experiment was run on a Core2Quad CPU at 2.4 GHz, using a single core. The memory usage was limited to 1 GB heap (to avoid swapping) and 400 MB stack (far more than the default 8 MB). A time limit is set to 30 minutes.

4.1 Experimental Setting

We have run experiments with all algorithms, always including NC as a pre-processor before any solver. The experiments we report rely on 100 randomly generated sets of n tasks (here $n = 10$ or 16). Associating each set with $m = 1$ to $n - 1$ processors, we obtain a total of $100.(n - 1)$ problems. Each task τ_i is randomly created by uniformly sampling: first $D_i \in [1..T_{\max}]$, then $C_i \in [1..D_i]$ and $T_i \in [D_i..T_{\max}]$, where $T_{\max} = 13$. Despite the possibly large hyperperiod ($T^H_{\max} = 360360$), it is expected that most of these problems will be easily solved either because they are obviously unfeasible scheduling problems, or because there are few conflicts for accessing to the various resources (the processors).

Tables 1 and 2 give a statistical summary of the main experiments for $n = 10$ and $n = 16$ tasks through the number of instances (i) proved unfeasible, (ii) proved feasible, (iii) unsolved, in particular (iv) due to a lack of time (column "~~time~~") and (v) due to a lack of memory (column "~~mem~~"), plus (vi) the average running time. Figure 3 also presents, for a selection of all tested algorithms, three plots giving, as a function of the number of processors, the percentages of instances: (i) proved unfeasible, (ii) proved feasible, or (iii) unsolved.

[3] Note that heuristic $u = 5$ is always used with the slack constraint.

Algorithm	unf.	feas.	unk.	~~time~~	~~mem~~	$E[time]$
NC	360	0	540	0	0	0±0
NS(0)	360	279	261	0	0	0±0
NS(1)	360	327	213	0	0	0±0
NS(2)	360	354	186	0	0	0±0
NS(3)	360	318	222	0	0	0±0
NS(4)	360	467	73	0	0	0±0
FPS(4)	360	478	62	1	0	6±7
CSP2(4)	360	514	26	26	0	54±87
CSP2(4)r	360	527	13	13	0	32±51
CSP2(4)s	362	528	10	10	0	20±39
CSP2(5)s	362	528	10	10	0	20±35
CSP2(4)sr	368	529	3	3	0	8±16
CSP2(5)sr	369	529	2	2	0	7±13

Table 1: Statistical summary of experiments for $n = 10$ tasks.

Algorithm	unf.	feas.	unk.	~~time~~	~~mem~~	$E[time]$
NC	597	0	903	0	0	0±0
NS(0)	597	514	389	0	0	0±0
NS(1)	597	564	339	0	0	0±0
NS(2)	597	621	282	0	0	0±0
NS(3)	597	551	352	0	0	0±0
NS(4)	597	801	102	0	0	0±0
CSP2(4)	597	863	40	40	0	51±94
CSP2(4)r	597	876	27	27	0	37±71
CSP2(4)s	597	896	7	7	0	8±19
CSP2(5)s	597	898	5	5	0	7±18
CSP2(4)sr	600	898	2	2	0	4±10
CSP2(5)sr	600	897	3	3	0	4±11

Table 2: Statistical summary of experiments for $n = 16$ tasks.

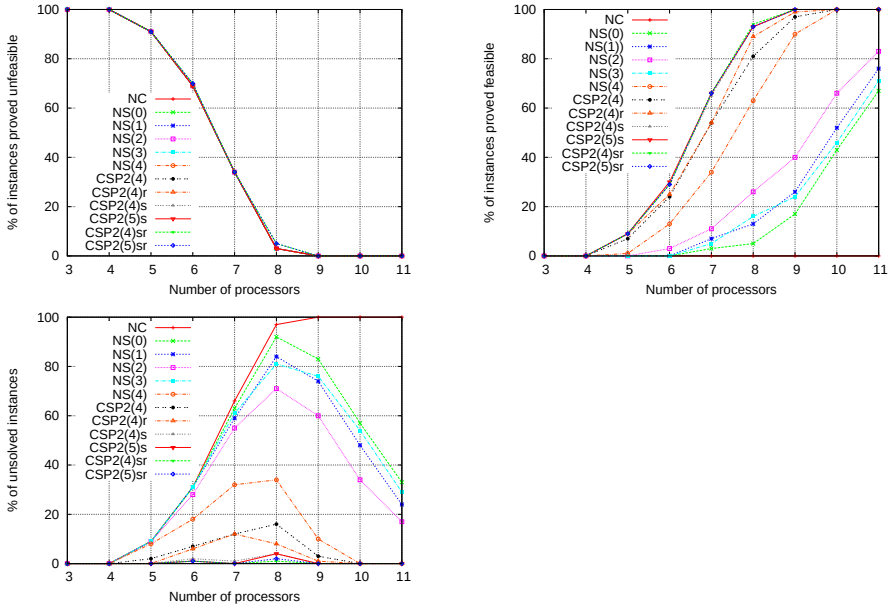

Figure 3: Results (probability of failure, success and undetermined outcomes) for $n = 16$ tasks and $m = 3..11$ processors.

4.2 Analysis

First, NC appears to identify almost all unfeasible instances (see Fig. 3). This is all the more important that proving a problem unfeasible is often a time consuming task. Thus, the various solvers are essentially left with the task of proving problems feasible.

NS is another simple and efficient algorithm as it is able to find a schedule for 50% to 88% of feasible instances. Heuristic $u = 4$ $(D - C)$ clearly outperforms other FP heuristics ($u = 0..3$), what has also been observed with other algorithms (results not reported). For convenience, instances not solved by NS(4) will be referred to as "hard instances".

FPS is a more evolved algorithm than NS. Yet it does not improve results significantly when using $u = 4$. In the $n = 10$ tasks case, it is typically very fast (less than 1 second) when finding a solution, but spends an average 61 seconds covering its search space. Because FPS(4) does not improve much on NS(4) but suffers from the combinatorial explosion of the search space, we did not conduct full experiments in the $n = 16$ tasks case ($\sim 2 \times 10^{13}$ states).

We also do not report experiments conducted with CSP1. Even with $n = 10$ tasks, CSP1 typically runs either out of memory or out of time (in these experiments, no other algorithm ran out of memory).

CSP2, as FPS, typically spends a lot of time on unfeasible instances (rarely on feasible ones). Yet, this is counterweighted by the fact that CSP2 (with $u = 4$ or 5) improves on NS and FPS by proving at least half of the "hard instances" feasible. In these same experiments, one does not observe any statistically significant difference between the ordering heuristics $u = 4$ and $u = 5$. Having to re-order tasks at each time step—when $u = 5$—does not seem to slow down CSP2.

As expected, monitoring the slack time or focusing on bottlenecks make for a significantly better solver (leaving at most one quarter of hard instances unsolved). A difference is that monitoring the slack time tends to help more with feasible instances, while focusing on bottlenecks tends to help more with unfeasible instances. A combination of both solves almost all problem instances, showing that they are complementary improvements on CSP2.

One can also observe that the propotion of "hard instances" (over all generated instances) decreases when the number of tasks increases. This is also related to the highest average computation times when $n = 10$. This phenomenon reflects the fact that the phase transition between easily solved unfeasible and feasible problems (see Fig. 3). It does not contradict the fact that increasing the number of tasks makes it possible to create harder problems.

Another notable point regarding time is the usually large standard deviation. This is a consequence of the solving time being close to 0 on most instances, and equal to 1800 seconds on unsolved instances.

5 Discussion and Future Work

This work is a first step towards efficiently applying search techniques to MGRTS problems. Only simple heuristics and systematic search algorithms have been considered here. The empirical results show that a good approach is to combine simple heuristic solvers (NC and NS(4)) with a more complex one (CSP2(u)sr). In such a combination, NS(4) would speed up the search on easy feasible problems without degrading performances in other cases.

Searching for a fixed priority assignement (FPS) is not a satisfying approach: it is not as efficient as CSP2 on hard instances and does not improve over the simple heuristic NS(4) on easy problems.

A typical real-world problem is to minimize the number of processors required to handle a given set of n tasks (an important question for mass-produced embedded systems). This implies solving the hardest cases for a given set of tasks, justifying the search for better resolution techniques. The main contributions of this work are therefore the two improvements for CSP2: (1) using a laxity-based cosntraint to prune the search earlier, and (2) focusing the search on hard bottlenecks. These ideas are complementary as the former speeds up the search significantly, while the later concentrates the effort on an important set of variables. A difficulty is to deal with problems involving multiple bottlenecks. An important research direction is thus to find a way to perform a search focusing on multiple important areas. This probably implies using multiple searches joining each other to produce a complete solution.

One weakness of the various algorithms considered in this paper is that they all require storing a complete schedule in memory.[4] Indeed the size of the schedule grows linearly with the hyperperiod $T^H_{\max}$ which, in the worst case, grows exponentially with the maximum period $T_{\max}$. One should then either control the hyperperiod, or work with other search spaces (like FP assignments) using other testing means. An FP assignment can for example be tested by simulating executions, stopping either when a constraint is broken or when some state is encountered twice (meaning that the execution is cycling). Another reason for avoiding solutions in the form of expanded

[4]With $T_{\max} = 13$ and $m = 15$, we have a maximum of 5×10^6 variables, i.e., 130 times more than in the largest instance of the CSP-08 competition.

schedules is that they may not be appropriate on embedded systems due to memory constraints.

Because the time complexity also increases quickly with n, m and T_{max}, a second step would be to simply give up on systematic searches and consider local search algorithms instead, even if it will make it impossible to prove a problem unfeasible. A natural direction would be to look at dynamic priority assignments, i.e., assignments depending on various features of the immediate state of the system.

In a longer term, one of our objectives is to move from the usual deterministic setting—where worst-case execution times are considered—to probabilistic settings—where a probability distribution over execution times is known for each task τ_i. This is a very different problem that requires different approaches, for example based on a Markov decision process formulation, but it seems to be a natural prerequisite to first study the deterministic case.

References

[1] P. Baptiste, C. Le Pape, and W. Nuijten, *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*, Kluwer Academic Publishers, 2001.

[2] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, 'A categorization of real-time multiprocessor scheduling problems and algorithms', *Handbook of Scheduling*, (2005).

[3] L. Cucu and J. Goossens, 'Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors', *Proc. of the 11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'07)*, (2007).

[4] L. Cucu-Grosjean and O. Buffet, 'Global multiprocessor real-time scheduling as a constraint satisfaction problem', in *Proceedings of the ICPP'09 Workshop on Real-time systems on multicore platforms: Theory and Practice (XRTS'09)*, (2009).

[5] Davis, R.I. and Burns, A., 'A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems', Technical report, University of York, (2009).

[6] S.K. Dhall and C.L. Liu, 'On a real-time scheduling problem', *Operations Research*, **26**, 127–140, (1978).

[7] J. Goossens, S. Funk, and S. Baruah, 'EDF scheduling on multiprocessors: some (perhaps) counterintuitive observations', *Proc. of the 8th Int. Conf. on Real-Time Computing Systems and Applications*, (2002).

[8] V. Kumar, 'Algorithms for constraint-satisfaction problems: A survey', *AI magazine*, **13**(1), (1992).

[9] C.L. Liu, 'Scheduling algorithms for multiprocessors in a hard real-time environment', *JPL Space Programs Summary*, **II**, 37–60, (1969).

[10] C.L. Liu and J.W. Layland, 'Scheduling algorithms for multiprogramming in a hard-real-time environment', *Journal of the ACM*, **20**(1), 46–61, (1973).

[11] A.M. Déplanche P.E. Hladik, H. Cambazard and N. Jussien, 'Solving a real-time allocation problem with constraint programming', *Journal of Systems and Software*, **81**(1), 132–149, (2008).

[12] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Englewood Cliffs, NJ: prentice Hall, 1995.

[13] Ernst S. Selmer, 'On the number of prime divisors of a binomial coefficient', *Math. Scand.*, **39**(2), 271–281, (1976).

[14] The Choco Team, 'Choco: An open source java constraint programming library', Technical report, Ecoles des Mines de Nantes, (2008). http://choco.emn.fr/.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399